VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace LDPC kódu v C++

LDPC implementation in C++

2014

Marek Fujaš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Marek Fujaš**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace LDPC kódu v C++
LDPC Implementation in C++

Zásady pro vypracování:

Cílem práce je integrovat LDPC kód do stávající optimalizační knihovny. Implementace může vycházet z dostupných otevřených implementací LDPC, zaměří se na integraci LDPC do optimalizační knihovny a přizpůsobení jejím potřebám. Součástí práce bude vyhodnocení efektivity implementace.

Seznam doporučené odborné literatury:

[1] V. Roca, C. Neumann, D. Furodet, ``Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes'', IETF RMT Working Group, RFC 5170 (Standards Track/Proposed Standard), June 2008.
[2] STMicroelectronics/AST and INRIA/Planète, ``LDPC benchmarking for DVB-H'', Document submitted to the DVB-H CDP Working Group, December 2005

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Pavel Krömer, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

„Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal."

23.7.2014

Abstrakt

Předmětem práce je implementace LDPC kódů v C++ a integrace do stávající knihovny. Písemná část obsahuje vysvětlení funkce a problematiky LDPC kódů, popis implementace LDPC kódů, a jejich integrace do sdílené knihovny AmphorA, použití v ukázkovém programu a testování v simulacích. Tyto kódy jsou využívány pro přenos zpráv zarušeným kanálem. Zpráva přijatá po přenosu takovým kanálem může obsahovat chyby. Typ těchto chyb závisí na typu kanálu. Pomocí LDPC kódů můžeme být schopni tyto chyby omezit, či úplně odstranit. Existuje několik typů LDPC kódů, způsobů jejich kódování a dekódování. Tato práce několik těchto typů uvádí a implementuje.

Klíčová slova: alist formát, BEC, Binary erasure kanál, Binary symetric kanál, Bit-flip kanál, BSC, Gallager, Generující matice, LDPC, Low Density Parity Check, McKay, Neal, Sum-product dekódování, Tannerův graf

Abstract

The subject of this work is the implementation of LDPC codes in C++ and integration in existing library. The document part explains fuctions and problems of LDPC codes, the description of implementation of LDPC codes, and its integration into AmphorA the shared library, demonstration in sample code, and benchmarking in simulations. These codes can be used for transfering messages through noisy channels. Message recieved through this channel could contain some errors. Type of these errors depends on the type of the channel. Using LDPC we might be able to reduce some of these errors, or even remove them completely. There are several types of LDPC codes, types of their encoding or decoding. In this work we'll be explaining some of these types and implementing them.

Keywords: alist format, BEC, Binary erasure channel, Binary symetric channel, Bit-flip channel, BSC, Gallager, Generator matice, LDPC, Low Density Parity Check, McKay, Neal, Sum-product decoding, Tanner graph

Obsah

1 Úvod

S chybou přenosu se setkáváme častěji, než si možná uvědomujeme. Několik hezkých příkladů jako jsou přenos bezdrátovou technologií, zápis na pevný disk, nebo dokonce přenos genetické informace uvedl ve své práci David McKay [2]. V této práci se ale budeme zabývat výhradně chybou v přenosu informace, která je zadána bitovým vektorem a její opravě pomocí tzv. Low Density Parity Check kódů. Probereme si kanály BEC a BSC, standardní LDPC kódování i kódování s úpravou na dolní trojúhelníkový tvar, dekódování na různých kanálech a dekódování soft-decision algoritmem Sum-product. Dále si popíšeme integraci LDPC kodéru a dekodéru do sdílené knihovny AmphorA, způsob jejich implementace, a použití v cílovém programu. Na konec budeme zkoušet efektivitu těchto algoritmů na simulovaných příkladech.

Při přenosu zprávy, se zde budeme setkávat s rušením, či šumem, jehož výsledkem se odeslaná zpráva znehodnotí podle pravidel daného přenosového kanálu. My se zde však nebudeme zabývat příčinou vzniku rušení, ať už se jedná o dorůstající strom na cestě bezdrátového spojení, či o vnější elektromagnetické vlivy způsobující chybný zápis na médium, budeme se zabývat způsoby jak zprávu upravit tak, aby po ji přenosu bylo možno rekonstruovat i přes vlivy rušení, konkrétně kódováním LDPC.

2 LDPC kódy

2.1 Přenosový kanál

Tyto kanály si rozdělíme podle toho, jak se chovají, dojde-li k rušení. Přenosové kanály se také dají dále dělit podle způsobu jakým rušení probíhá při přenosu signálem. Např. kanál AWGN, který přičítá rušivý signál k signálu zprávy.

2.1.1 BEC - Binary erasure channel

Jak si můžeme představit už podle názvu tohoto typu kanálu, bude docházet k smazání té části informace, která byla narušena. V cíli tedy obdržíme v každém bitu 0 nebo 1, pokud bit dorazil v pořádku. V opačném případě bude pak obsah bitu smazán (erasure). Tím se nabízí výrazné zjednodušení a zvýšení spolehlivosti dekodéru, neboť u informací obdržených tímto kanálem přesně víme, která část dorazila v pořádku, a kterou je třeba opravit dekódováním.

2.1.2 Bit-flipping kanály

Je kanál, ve kterém dojde-li k poškození nějakého bitu zasílané informace, změní poškozený bit svou hodnotu na opačnou. (0 se zamění za 1 a naopak). Tyto kanály jsou méně spolehlivé pro dekódování než BEC, protože musíme vždy prověřit celou zprávu, neboť nevíme zda je některá část špatně, ani která to bude. Navíc mohou nastat situace, kdy algoritmus, na základě šumem změněných okolních bitů „opraví“ i bit který jsme obdrželi neporušený.

2.2 Kódování

Uvažujeme-li přenos informace po přenosovém kanálu, je třeba počítat s tzv. šumem (noise), při kterém dochází část informace zkreslena. Je tedy třeba zajistit, aby se informace přečtená v cíli, shodovala s informací odeslanou zdrojem. Kromě vylepšení kvality přenosového kanálu můžeme využít systematického řešení – přenášenou informaci ve zdroji upravíme pomocí určitého kódu a v cíli ji opět rozkódujeme. Aby toto řešení mělo nějaký smysl, je zřejmé že zakódovaná zpráva musí být větší, než původní.

Pro příklad uvedeme libovolnou informaci **s**, kódovanou přenášenou informaci **t** a informaci přečtenou v cíli **r** zkreslenou šumem **n**. Kódování provedeme jednoduchým algoritmem, kdy každý bit zdrojové zprávy naklonujeme tak, že v kódované zprávě bude představovat 3 bity. Dekódování pak rozhodneme většinou ze tří bitů[2].

```
s =  0   0   1   0   1   1   0
t = 000 000 111 000 111 111 000
n = 000 001 000 000 101 000 000
```

```
r = 000 001 111 000 010 111 000
```

[1] Po dekódování tedy dostaneme tuto informaci: $\hat{s} = 0010010$ Jak je vidět tento algoritmus opravil šum ve druhém bitu informace (zleva), ale už neopravil chybu v šestém bitu. Ve větším měřítku bychom zjistili, že tento kód dosahuje značného zlepšení kvality, kterou lze dále vylepšit zvýšením počtu opakování každého bitu. Avšak zároveň je jasné, že přenos bude také mnohem pomalejší. V našem příkladě informace nabobtnala při přenosu do třikrát většího rozměru, než měla původní zpráva. Vzorec 1 ukazuje přesnější vztah mezi velikostí zprávy a zlepšení kvality u opakovacího kódu.

$$p_b = \sum_{n=(N+1)/2}^{N} \binom{N}{n} f^n (1-f)^{N-n} \tag{1}$$

kde p_b je pravděpodobnost chyby po rozkódování zprávy, N je počet opakování každého bitu a f je pravděpodobnost „otočení“ jednoho bitu. Tímto způsobem je možné minimalizovat následky šumu bohužel za cenu výrazného zatížení přenosového kanálu. Proto používáme složitější kódy, kde se poměr zlepšení kvality proti zatížení přenosového kanálu optimalizuje. Jiným příkladem může být např. Hammingův kód, v této práci se ale budeme nadále zajímat o kódy LDPC.

2.3 LDPC (Low-Density Parity-Check) kódy

Tyto kódy jsou pojmenované podle LDPC matic, které jsou využívány jako klíče pro kódování a dekódování. Byly poprvé navrženy v roce 1962 Robertem G. Gallagerem. V té době však jejich potenciál zůstal nevyužit kvůli nedostatečné síle tehdejší výpočetní techniky. Byly využity až mnohem později v 90. letech, kdy jimi McKay a Neal nahradili tzv. Turbo kódy.

2.3.1 LDPC matice

Každý LDPC kód je určen LDPC maticí. Tyto binární matice jsou typické nízkým počtem „jedniček“. Mohou být pravidelné – *regulární*, či nepravidelné *iregulární*. Díky nízkému

[1] Pokud bychom použili BEC kanál stačilo by přečíst vždy první nesmazaný bit z každé trojice. Při šumu n by pak zpráva dorazila v pořádku, problém by nastal až v momentě, kdy by byla šumem smazána celá trojice.

počtu jedniček se často využívá možnost zápisu, kdy místo celé matice opisujeme jen souřadnice jedniček a její rozměry (což je vzhledem k rozměrům v praxi využívaných matic jistě přehlednější).

2.3.1.1 Regulární LDPC matice LDPC matice (w_c, w_r) se nazývá regulární, jestliže každý kódovací bit je obsažen v čísle w_c a počet parity–check rovnic je roven w_r . Jinak také počet jedniček ve sloupci je roven w_c ,a počet jedniček v řádku je roven w_r . Gallager definoval regulární LDPC matici složenou z *j* podmatic, kdy každá část má stejný počet sloupců, v každém sloupci je jedna jednička. První z těchto podmatic má jedničky vždy schodově sestupně řazené. Sloupce ostatních podmatic jsou pak permutacemi sloupce v první podmatici[3]. Příkladem takové matice může být matice 2 s $w_c = 4, w_r = 3, M = 15, N = 20$, kde M a N určují rozměry matice.

$$
[3]\quad
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}
\qquad (2)
$$

Dnes používáme regulární matice, které jsou dány pouze svou délkou a uniformitou váhy řádků a sloupců. Příkladem délky 12 (3,4)regulární LDPC matice může být matice 3

.

$$
\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1
\end{array}
\qquad (3)
$$

V tomto případě doplňujeme pouze sloupce zleva doprava přičemž v každém sloupci

dodržujeme jeho váhu podle w_c. Přednostně vybíráme řádky s nejnižší vahou. Pokud není dosažena rovnováha $w_c a w_r$ v celé matici (degree distribution) měníme postupně sloupce, dokud se nepoměr neodstraní. Používají se též matice sloupcově regulární, které nemají konstantní váhu řádků, pouze konstantní váhu sloupců.

2.3.1.2 Iregulární matice Jsou ty LDPC matice, kde distribuce jedniček v řádcích, či sloupcích není konstantní[1]. Příkladem může být opakovaně-akumulativní matice (kód) (repeat-acumulaticve code). Jejíž část tvoří sestupné schody, každý s počtem $w_c = 2$, která je délky M. Jedna taková matice délky 12 a poměru 1:4 následuje na 4.

$$[1] \quad \begin{matrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{matrix} \tag{4}$$

2.4 LDPC Kodér

2.4.1 Standardní kódování

Pro kódování zpráv je třeba nejprve sestavit takzvanou generující matici (generator matrix) G. Standardní způsob k jejímu vytvoření je úprava LDPC matice H o počtu sloupců n a počtu řádků k na schodový tvar[1]. Tedy

$$H = [A, I_{n-k}] \tag{5}$$

, kde A je binární matice o počtu řádků k a počtu sloupců $(n - k)$ Odtud potom získáme generující matici jako

$$[1] \quad G = [I_k, A^T] \tag{6}$$

Příklad si demonstrujeme na následující matici H.

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{7}$$

Pomocí elementárních řádkových operací (vzájemná výměna řádků a součet řádků modulo 2) dostaneme matici H, která obsahuje jednotkovou podmatici o počtu řádků k. Prohozením 3. a 5. řádku, ke 4. řádku přičteme 1. řádek, a k 5. přičteme 4. řádek. Dostaneme matici, která má v levou část v horním trojúhelníkovém tvaru. Když potom přičteme k 1.

řádku 2., 3., 4. a 5. řádek, ke 2. řádku přičteme 3. a 5. řádek, a ke 4. řádku přičteme 5. řádek, dostáváme konečně matici ve tvaru $H = [A, I_{n-k}]$,kterou nazveme H_{rr} pro pozdější účely,

$$H_{rr} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

odtud pak získáme $G = [I_k, A^T]$

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{9}$$

Odtud pak můžeme generovat kódovanou zprávu *c* z původní zprávy *u* takto:

$$c = u \cdot G \tag{10}$$

Tedy například ze zprávy $u = 11001$ získáme

$$c = 1100111011 \tag{11}$$

Tento způsob vytváření matice G bohužel mění matici G způsobem, kdy přestává být "řídkou"(sparse). Díky tomu pak narůstá složitost operace samotného kódování, která se tak blíží n^2. Proto si uvedeme ještě jeden způsob kódování jehož složitost je vždy téměř lineární. V tomto případě místo hledání generující matice převedeme H do horní **trojúhelníkové** formy[1]. Přitom použijeme dokud to bude možné jen řádkové a sloupcové permutace, abychom zajistili, že H zůstane řídká. Nejprve tedy dostaneme matici H do tvaru

$$[1] \quad G = \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix} \tag{12}$$

,kde T je dolní trojúhelníková matice o rozměrech $(m-g) \times (m-g)$, má-li matice H poměr řádků a sloupců 1/2, pak je A o velikosti $(m-g) \times k$, a B je o velikosti $(m-g) \times g$, g je počet řádků v C,D a E ,a nazývá se mezerou (gap) reprezentace. Obecně platí, že čím menší je mezera, tím menší je složitost kódování. Nyní si převedeme matici H z 7 do tohoto tvaru prohozením druhého a třetího řádku, a šestého a desátého sloupce.

$$H_T = \left(\begin{array}{ccccc|cc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \tag{13}$$

,kde $g = 2$. Dalším krokem bude vynulování podmatice E:

$$[1]\ \ \tilde{H} = \begin{pmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{pmatrix} H_T = \begin{pmatrix} A & B & T \\ \tilde{C} & \tilde{D} & 0 \end{pmatrix} \tag{14}$$

, odkud

$$[1]\ \ \tilde{C} = -ET^{-1}A + C \tag{15}$$

a

$$[1]\ \ \tilde{D} = -ET^{-1}B + D \tag{16}$$

V našem příkladu dostaneme:

$$\tilde{H} = \left(\begin{array}{ccccc|cc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right) \tag{17}$$

Pro kódovanou zprávu $c = [u, p_1, p_2]$ musí platit [1] $c\tilde{H}^T = 0$ a tedy

$$[1]\ \ Au + Bp_1 + Tp_2 = 0\tilde{C}u + \tilde{D}p_1 + 0p_2 = 0 \tag{18}$$

, odtud

$$[1]\ \ p_1 = \tilde{D}^{-1}\tilde{C}u \tag{19}$$

$$[1]\ \ p_2 = -T^{-1}(Au + Bp_1) \tag{20}$$

Můžeme nyní zkusit zakódovat znovu stejnou zprávu $u = 11001$ stejnou maticí H. Zjistíme že $p_1 = 10$ a $p_2 = 100$, potom

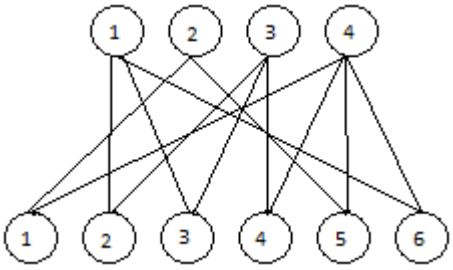
$$c = 1100110100 \tag{21}$$

2.5 Tannerův graf

Tannerův graf je tvořen dvěma typy vrcholů. Jsou to bitové vrcholy (bit nodes), jejichž počet je roven počtu sloupců LDPC matice a kontrolní vrcholy (check nodes), jejichž počet je roven počtu řádků LDPC matice. Hranu mezi těmito vrcholy pak vytvoříme vždy, když má řádek s číslem stejným jako je číslo daného kontrolního vrcholu jedničku ve sloupci, jehož číslo je stejné jako je číslo daného bitového vrcholu. Hrany mohou existovat jen

mezi bitovým a kontrolním vrcholem. Vznikne-li v grafu cyklus, pak se nejedná o LDPC matici[1]. Jako příklad si vytvoříme Tannerův graf pro následující matici 22.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{22}$$



Obrázek 1: Tannerův graf pro matici 22.

2.6 BEC hard-decision dekodér

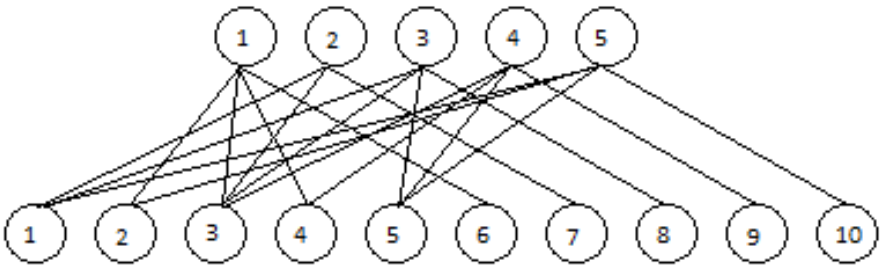
Pro příklad dekódujeme zprávu z příkladu 11 $c = [1100111011]$ Nyní vyrušíme několik bitů $m = [1x00111xx1]$, kde x je smazaný bit zprávy. Pro dekódování je třeba získat z matice H matici H_{std} permutací sloupců matice H_{rr}, kde posledních m řádků je shodných s m prvními řádky matice H_{rr}.

$$H_{std} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{23}$$

Touto maticí budeme zprávu dekódovat, a pro lepší názornost si vytvoříme Tannerův graf pro tuto matici: Protože musí platit pravidlo, že součet kontrolních vrcholů, které jsou spojeny hranou s daným bitovým vrcholem musí být roven 0, můžeme vytvořit rovnice, které dekódují vymazané bity [1]. Pro první bitový vrchol například platí:

$$c_2 + c_3 + c_4 + c_6 = 0x + 0 + 0 + 1 = 0 \tag{24}$$

‚tedy x je v tomto případě 1, a ze zprávy m nyní máme: $m = 1100111xx1$ Teď bychom prakticky mohli skončit s dekódováním, protože kódované slovo, které tvoří první polovinu zprávy už bezpečně máme, ale pro názornost dekódujeme i zbytek zprávy. Podle 3.



Obrázek 2: Tannerův graf pro matici 23.

a 4. bitového vrcholu:

$$B_3 : 1 + 0 + 1 + x = 0$$
$$B_4 : 0 + 0 + 1 + x = 0$$

a konečně máme celou zprávu $m = c = 1100111011$. V tomto případě došlo k bezchybnému rozkódování zprávy při přibližném šumu 33%. Pochopitelně by mohla nastat situace, kdy nebude vzhledem k výšce šumu možné výchozí zprávu jednoznačně určit, na rozdíl od bit-flip kanálu bychom ale věděli přesně že tato situace nastala.

2.7 Bit-flip hard-decision dekodér

[1] Pro bit-flip dekódování použijeme jako příklad 21. Matice ve tvaru H_t poslouží pro dekódování zprávy $c = 1100110100$, ve které přehodíme například devátý bit, a získáme tak zprávu $m = 1100110110$. Dekódování bude teď probíhat v jednotlivých iteracích. Budeme iterovat jednotlivé rovnice kontrolních vrcholů.

$$C_1(B_1, B_2, B_4, B_5, B_8) : 1 + 1 + 0 + 1 + 1 = 0$$
$$C_2(B_4, B_6, B_8, B_9) : 0 + 1 + 1 + 1 = 1$$
$$C_3(B_2, B_3, B_5, B_7, B_10) : 1 + 0 + 1 + 0 + 0 = 0$$
$$C_4(B_1, B_2, B_7, B_9, B_10) : 1 + 1 + 0 + 1 + 0 = 1$$
$$C_5(B_3, B_6, B_8, B_10) : 0 + 1 + 1 + 0 = 0$$

Z faktu že součty nevycházejí sudě poznáme, že zpráva došla poškozená. Určíme hodnotu jednotlivých bitů zprávy z těchto rovnic.

$$E_{1,1} = 1+0+1+1 = 1$$
$$E_{1,2} = 1+0+1+1 = 1$$
$$E_{1,4} = 1+1+1+1 = 0$$
$$E_{1,5} = 1+1+0+1 = 1$$
$$E_{1,8} = 1+1+0+1 = 1$$

$$E_{2,4} = 1+1+1 = 1$$
$$E_{2,6} = 0+1+1 = 0$$
$$E_{2,8} = 0+1+1 = 0$$
$$E_{2,9} = 0+1+1 = 0$$

$$E_{3,2} = 0+1+0+0 = 1$$
$$E_{3,3} = 1+1+0+0 = 0$$
$$E_{3,5} = 1+0+0+0 = 1$$
$$E_{3,7} = 1+0+1+0 = 0$$
$$E_{3,10} = 1+0+1+0 = 0$$

$$E_{4,1} = 1+0+1+0 = 0$$
$$E_{4,2} = 1+0+1+0 = 0$$
$$E_{4,7} = 1+1+1+0 = 1$$
$$E_{4,9} = 1+1+0+0 = 0$$
$$E_{4,10} = 1+1+0+1 = 1$$

$$E_{5,3} = 1+1+0 = 0$$
$$E_{5,6} = 0+1+0 = 1$$
$$E_{5,8} = 0+1+0 = 1$$
$$E_{5,10} = 0+1+1 = 0$$

Jak je vidět, ve všech rovnicích kontrolních vrcholů, které vyšly liché se nyní hodnoty všech bitů otočily a naopak, a určíme bit, který změníme podle většiny. Například první bit zprávy má dva kontrolní vrcholy pro 1 a žádný pro 0. $m = 1100110100$ Další kontrolou zjistíme, že součet na všech kontrolních vrcholech je roven 0, a cyklus skončí. Výsledná zpráva **m** je v tomto případě rovna odeslané zprávě **c**. Problémem tohoto takzvaného hard-decision algoritmu jsou cykly uvnitř Tannerova grafu. Totiž s každým otočeným bitem přijaté zprávy se otočí výsledek všech bitů na kontrolních vrcholech majících hranu s tímto bitem. Je-li pak nějaký bit obsažen většinou na těchto vrcholech, bude také změněn, což v případě, že byl přijat správně, může vyústit v další poškození zprávy. Může dokonce nastat situace, kdy se výsledky iterací cyklicky opakují.

2.7.1 Dekódování sumou a produktem

Dekódování sumou a produktem (Sum-Product Decoding) je ukázkou soft-decision algoritmu na bit-flip kanálu. My si jej demonstrujeme znovu na příkladu 21. Odeslanou zprávou bude opět $c = 1100110100$ a obdrženou zprávou bude $m = 1000110100$. Tento způsob dekódování vyžaduje určení přibližné pravděpodobnosti otočení bitu. V našem příkladu si stanovíme tuto pravděpodobnost $p = 0,1$, což navíc odpovídá faktu, že jsme přehodili jen jeden bit odeslané zprávy. Tento algoritmus využívá při rozhodování věrohodnosti (likelihood), tedy každý z výsledků rovnic kontrolních vrcholů vykazuje jistou míru věrohodnosti, na rozdíl od předchozího algoritmu, kde všechny výsledky měly stejnou váhu. V našem případě je věrohodnost pro každý bit zprávy:

$$[1]\ \ log\left(\frac{p}{1-p}\right) = log\left(\frac{0,1}{0,9}\right) = -2.197224577 \text{ , pro 1} \tag{25}$$

$$log\left(\frac{1-p}{p}\right) = log\left(\frac{0,9}{0,1}\right) = 2.197224577 \text{ , pro } 0 \tag{26}$$

Potom věrohodnost jednotlivých bitů zprávy můžeme znázornit vektorem
$r = [-2.197, 2.197, 2.197, 2.197, -2.197, -2.197, 2.197, -2.197, 2.1972.197]$

Nyní znovu sestavíme z rovnic kontrolních vrcholů hodnoty pro jednotlivé bity zprávy podle vzorce 27.

$$[1] \quad E_{k,b} = log\left(\frac{1+\prod_{i'\in B_k, i'\neq i} tanh(M_{k,i'}/2)}{1-\prod_{i'\in B_k, i'\neq i} tanh(M_{k,i'}/2)}\right) \tag{27}$$

, kde **(**k) je číslo kontrolního vrcholu, **B** je množina bitových vrcholů napojených na kontrolní vrchol **k** a **b** je číslo bitového vrcholu, čili číslo bitu zprávy a $M_{k,i} = r_i$. A z výsledků těchto rovnic vypočítáme jednotlivá L_i vyhodnocení hodnot jednotlivých bitů zprávy.

$$E_{1,1} = log\left(\frac{1+tanh(M_{1,2}/2)tanh(M_{1,4}/2)tanh(M_{1,5}/2)tanh(M_{1,8}/2)}{1-tanh(M_{1,2}/2)tanh(M_{1,4}/2)tanh(M_{1,5}/2)tanh(M_{1,8}/2)}\right) =$$
$$log\left(\frac{1+2.197\cdot 2.197\cdot(-2.197)\cdot(-2.197)}{1-2.197\cdot 2.197\cdot(-2.197)\cdot(-2.197)}\right) = 0,870$$
$$E_{1,2} = log\left(\frac{1+tanh(M_{1,1}/2)tanh(M_{1,4}/2)tanh(M_{1,5}/2)tanh(M_{1,8}/2)}{1-tanh(M_{1,1}/2)tanh(M_{1,4}/2)tanh(M_{1,5}/2)tanh(M_{1,8}/2)}\right) =$$
$$log\left(\frac{1+(-2.197)\cdot 2.197\cdot(-2.197)\cdot(-2.197)}{1-(-2.197)\cdot 2.197\cdot(-2.197)\cdot(-2.197)}\right) = -0,870$$

$$E_{1,4} = \ldots = -0,870$$
$$E_{1,5} = \ldots = 0,870$$
$$E_{1,8} = \ldots = 0,870$$
$$E_{2,4} = \ldots = 1,130$$
$$E_{2,6} = \ldots = -1,130$$
$$E_{2,8} = \ldots = -1,130$$
$$E_{2,9} = \ldots = 1,130$$
$$E_{3,2} = \ldots = -0,870$$
$$E_{3,3} = \ldots = -0,870$$
$$E_{3,5} = \ldots = 0,870$$

$$E_{3,7} = \ldots = -0,870$$
$$E_{3,10} = \ldots = -0,870$$
$$E_{4,1} = \ldots = 0,870$$
$$E_{4,2} = \ldots = -0,870$$
$$E_{4,7} = \ldots = -0,870$$
$$E_{4,9} = \ldots = -0,870$$
$$E_{4,10} = \ldots = -0,870$$
$$E_{5,3} = \ldots = 1,130$$
$$E_{5,6} = \ldots = -1,130$$
$$E_{5,8} = \ldots = -1,130$$
$$E_{5,10} = \ldots = 1,130$$

$$L_1 = l_1 + E_{1,1} + E_{4,1} =$$
$$-2.197 + 0,870 + 0,870 = -0,456$$
$$L_2 = l_2 + E_{1,2} + E_{3,2} + E_{4,2} =$$
$$2.197 - 0,870 - 0,870 - 0,870 = -0.413$$
$$L_3 = l_3 + E_{3,3} + E_{5,3} =$$
$$2.197 - 0,870 + 1,130 = 2.457$$
$$L_4 = l_4 + E_{1,4} + E_{2,4} =$$
$$2.197 - 0,870 + 1,130 = 2.457$$
$$L_5 = l_5 + E_{1,5} + E_{3,5} =$$
$$-2.197 + 0,870 + 0,870 = -0.456$$

$$L_6 = l_6 + E_{2,6} + E_{5,6} =$$
$$-2.197 - 1,130 - 1,130 = -4.458$$
$$L_7 = l_7 + E_{3,7} + E_{4,7} =$$
$$2.197 - 0,870 - 0,870 = 0.456$$
$$L_8 = l_8 + E_{1,8} + E_{2,8} + E_{5,8} =$$
$$-2.197 + 0,870 - 1,130 - 1,130 = -3.588$$
$$L_9 = l_9 + E_{2,9} + E_{4,9} =$$
$$2.197 + 1,130 - 0,870 = 2.457$$
$$L_{10} = l_{10} + E_{3,10} + E_{4,10} + E_{5,10} =$$
$$2.197 - 0,870 - 0,870 + 1,130 = 1.587$$

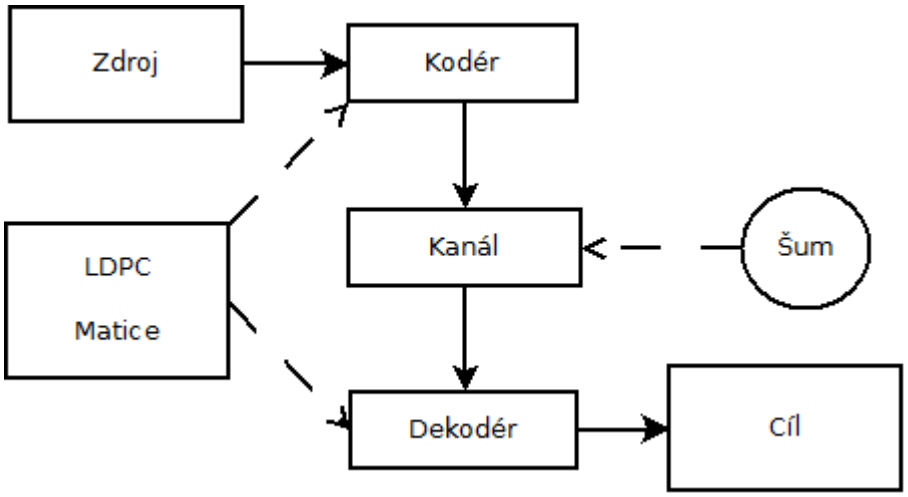
, kde l_i jsou věrohodnosti jednotlivých bitů obdržené zprávy. Pro samotné dekódování pak každý bit M_i dekódované zprávy m se určí podle znaménka L_i, kdy $M_i = 1$ je-li L_i záporné a $M_i = 0$ je-li L_i kladné.

2.8 Shrnutí LPDC kódů

LDPC kódy jsou určeny maticí, z níž vzniká kodér a dekodér, a způsobem úpravy této matice. Kodér samotný využívá pouze algoritmus spojený se způsobem tvorby matice, zatímco dekodér může pracovat s větším množstvím algoritmů, a musí rozeznávat různé typy kanálů, protože každý vyžaduje jiný přístup k dekódování. V následující části práce bude vysvětlena implementace kodéru, dekodéru a dalších částí, potřebných pro jejich funkčnost. Byla implementována tvorba matice jak standardním způsobem, tak pomocí dolní trojúhelníkové formy. Implementace dekodéru obsahuje tři výše zmíněné algoritmy, tedy hard-decision algoritmus pro BEC a bit-flip kanál a Sum-product algoritmus pro bit-flip kanál. Tvorba LDPC matic není součástí implementace, součástí je pouze jejich úprava pro kodér a dekodér.

3 Popis implementace LDPC kódů

V této části si popíšeme implementaci LDPC algoritmů pro tvorbu kódů, kódování a dekódování zpráv a jejich využití v ukázkovém programu. Implementace kódů LDPC je součástí sdílené knihovny AmphorA. Tato knihovna spolu s ukázkovým programem, který demonstruje použití implementace LDPC jsou obsaženy v příloze. Knihovnu sestavíme pomocí makefile v adresáři *AmphorA/target/* příkazem *make all*. Vzniklá knihovna **libAmphorA.so** se pak může linkovat s cílovým programem.
LDPC hlavičkové soubory jsou umístěny v knihovně v **arg/utils/ldpc**. Implementované části si ukážeme na náčrtu 3.

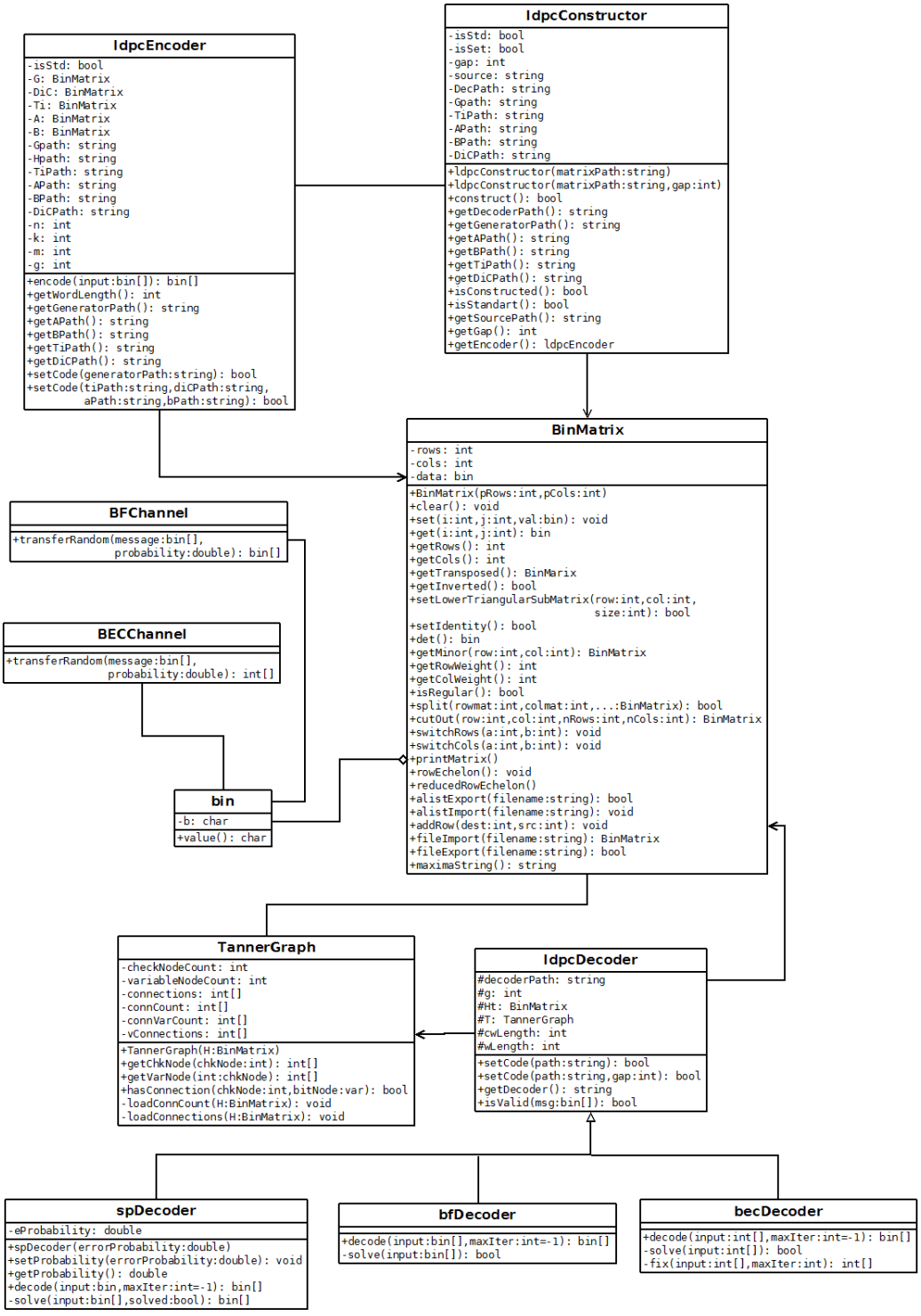


Obrázek 3: Koncept LDPC kodéru a dekodéru

Implementace se zaměřuje především na LDPC kodér a dekodér, pro tvorbu těchto prvků bylo třeba implementovat i tvorbu LDPC kódů, a pro simulaci byly přidány jednoduché přenosové kanály.

3.1 Popis tříd a metod

Způsob implementace načrtneme třídním diagramem 3.1.



Obrázek 4: Třídní diagram.

Kodér i dekodér lze zkonstruovat pouze z předem upravených matic v alist formátu. Tyto matice lze vytvořit pomocí třídy ldpcConstructor, která vytvoří z LDPC matice, kód pro kodér i dekodér. Důvodem proč nepředáváme rovnou celé matice je fakt, že LDPC matice bývají rozsáhlé ,a proto je lepší ukládat zdlouhavý proces úpravy kódu pro kodér a dekodér do souboru, abychom nemuseli provádět zbytečně tytéž operace stále dokola. Výsledkem metody construct třídy ldpcConstructor jsou tedy soubory s příponami (.g, .dec, .a, atd.), jejichž význam si popíšeme dále. Pro simulaci zarušení zprávy jsou k dispozici třídy BECChannel a BFChannel, které za použití standardní knihovny jazyka C (funkce rand), náhodně zaruší zprávu.
Následuje popis těchto tříd, jejich metod a fukncí stojících mimo tyto třídy.

3.1.1 Datový typ bin

Základ implementace tvoří třída **bin** deklarová v hlavičkovém souboru **bin.h**. Deklaruje třídu **bin** jako jeden bit, její chování v operátorech sčítání, násobení, negace a další, a konverze se standardními datovými typy.
Příklad použití:

```
bin a;          // Vytvoření proměnné
bin a = 0;      // Vytvoření proměnné s přiřazením hodnoty
bin b = 1;      // Vytvoření proměnné s přiřazením hodnoty
bin c = a + b; // XOR  (výsledkem je 1)
c = !a;         // NOT  (výsledkem je 1)
c = a * b;      // AND  (výsledkem je 0)
c = a / b;      // OR   (výsledkem je 1)
```

3.1.2 Formát alist

Tento formát byl využíván pro zápis matic Davidem MacKayem, Matthew Daveym, a Johnem Laffertym, jedná se úsporný, ale obsáhlý zápis LDPC mtice. Protože tyto matice jsou řídké, stačí zapisovat pouze pozice jedniček. Navíc abychom ukrátili čas tvorby Tannerova grafu se v alistu zaznamenávají váhy řádků i sloupců a pozice jedniček jak v jednotlivých sloupcích, tak i v jednotlivých řádcích. Ukázkou takového zápisu matice je následující kód a matice kterou reprezentuje.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

```
10 5
2 4
2 2 2 2 2 2 2 2 2 2
4 4 4 4 4
1 2
1 3
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
1 2 3 4
1 5 6 7
2 5 8 9
3 6 8 10
4 7 9 10
```

Význam řádků alist formátu je následující:

1. řádek obsahuje počet sloupců(m) a počet řádků(n)

2. řádek obsahuje maximální váhu sloupce a maximální váhu řádku

3. řádek obsahuje m čísel, každé reprezentuje váhu jednoho sloupce

4. řádek obsahuje n čísel, každé reprezentuje váhu jednoho řádku

5. až (m + 4). řádek obsahuje pozice jedniček daného sloupce

(m + 5). řádek a dále obsahuje pozice jedniček daného řádku

3.1.3 TannerGraph

TannerGraph je třída deklarovaná hlavičkovým souborem **tannergraph.h**, která reprezentuje Tannerův graf využívaný dekodérem.

```
TannerGraph(BinMatrix *H)
```

Konstruktor který vytvoří Tannerův graf podle matice H.

```
std::vector<int> getChkNode(int node)
std::vector<int> getVarNode(int node)
```

Vrací vektor všech vrcholů spojených hranou s daným kontrolním, či bitovým vrcholem.

3.1.4 ldpcConstructor

Pro urychlení práce s kodérem a dekodérem jsou při procesu úpravy LDPC matice na generující matici všechny výsledky ukládány do souborů ve formátu alist. Protože u obsáhlých matic může tento proces trvat až několik minut, je vhodnější použít matice kodérem už upravené pro ušetření času. Tyto maticové souboru jsou generovány do adresáře, ve kterém se nachází zdrojová LDPC matice,a jejich názvem je název matice ukončený specifickou příponou. Tvorbu těchro souborů implementuje třída ldpcConstructor z hlavičkového souboru **ldpcconstructor.h**.

```
ldpcConstructor(std::string matrixPath,int gap)
```

Konstruktor, pro vytvoření LDPC konstruktoru pro kódy vytvářené trojúhelníkovou úpravou.

```
ldpcConstructor(std::string matrixPath)
```

Konstruktor pro vytvoření standardního LDPC konstruktoru.

```
bool construct();
```

Provede úpravu LDPC kódu a vytvoří soubory pro kodér a dekodér.

```
bool construct();
```

Provede úpravu LDPC kódu a vytvoří soubory pro kodér a dekodér.

```
std::string getDecoderPath()
std::string getGeneratorPath()
std::string getAPath()
std::string getBPath()
std::string getTiPath()
std::string getDiCPath()
std::string getSourcePath()
```

Pro získání cest k jednotlivým vytvořeným maticím.

```
ldpcEncoder getEncoder()
```

Je-li kód zkonstruován, vrátí LDPC kodér s tímto kódem.

3.1.4.1 Generované soubory

U standardního kódu vznikají soubory s příponou **.dec** a **.g** pro dekodér a generátor, u kódu vzniklého trojúhelníkovou úpravou jsou to soubory **.a**, **.b**, **.ti**, **.dic**, a **.dec**. Podmatice A,B,C,D a T odpovídají označení podmatic v příkladu 12. Soubory generované kodérem:

Přípona souboru	Popis matice
.G	Generující matice standardního kodéru.
.a	Je levou horní částí generující matice využívající dolní trojúhelníkový tvar, výše označenou jako A.
.b	Je střední horní částí generující matice využívající dolní trojúhelníkový tvar, výše označenou jako B.
.ti	Je invertovanou pravou horní částí generující matice využívající dolní trojúhelníkový tvar, výše označenou jako T.
.dic	Je upravenou částí generující matice využívající dolní trojúhelníkový tvar. Vznikne vynásobením invertované střední dolní části levou dolní částí matice, výše označených jako C a D.
.dec	Je matice pro dekódování zprávy. Tuto lze použít v dekodéru a vyhnout se tak opětovnému upravování zdrojové LDPC matice.

3.1.5 ldpcEncoder

ldpcEncoder je třída deklarovaná hlavičkovým souborem **ldpcencoder.h**, která reprezentuje LDPC kodér. Instance této třídy je schopna po zadání kódu kódovat zprávy pro přenos a pozdější rozkódování dekodérem.

```
bool setCode(std::string generatorPath)
bool setCode(std::string tiPath, std::string diCPath,
std::string aPath, std::string bPath)
```

Pro nahrání LDPC kódu předem upraveného pro kodér. Nastaví kód ze souboru ,či souborů určených parametry.

```
std::vector<bin> encode(std::vector<bin> input)
```

Zakóduje danou zprávu, pokud je na kodéru nastaven kód. Pracuje se zprávami jako s vektory typu **bin**.

```
int getWordLength()
```

Vrací délku zprávy, kterou lze přijmout kodérem.

```
std::string getGeneratorPath()
std::string getAPath()
std::string getBPath()
std::string getTiPath()
std::string getDiCPath()
```

Pro získání cest k maticím které byly kodéru předány. (prázdný řetězec, tam kde nastaveny nejsou)

3.1.6 ldpcDecoder

ldpcDecoder je třída deklarovaná hlavičkovým souborem **ldpcdecoder.h**, která reprezentuje obecný LDPC dekodér. Podobně jako ldpcEncoder i tato třída pracuje s upravenou maticí v souboru ve formátu alist. Tato třída neobsahuje žádný dekódovací algoritmus, pouze uchovává kód.

```
bool setCode(std::string path)
bool setCode(std::string path, int gap)
```

Pro nahrání kódu předem upraveného pro dekodér.

3.1.7 becDecoder

becDecoder je třída deklarovaná hlavičkovým souborem **ldpcdecoder.h**, která reprezentuje LDPC dekodér na BEC kanálu. Instance této třídy je schopna po zadání kódu dekódovat zprávy přijaté po přenosu BEC kanálem a opravovat v nich chyby vzniklé šumem.

```
std::vector<bin> decode(std::vector<int> input,
const int maxIter = NOTSET)
```

Slouží k dekódování kódované zprávy zprávy, která je vektorem typů int, protože BEC kanál připouští po přijetí zprávy tři možnosti pro každý bit zprávy. Bity které obsahují 1, nebo 0 jsou přijaty v pořádku, bity které obsahují cokoli jiného jsou považovány za smazané bity. Zpět vrací binární vektor, takže pokud se nepodařilo během dekódování opravit všechny smazané bity, budou tyto vyplněny pseudonáhodnou hodnotou. Hodnota maxIter je-li nastavena udává maximální počet iterací, kterými proces dekódování může projít. Dekódování skončí buď tím, že zpráva byla celá opravena, nebo tím že poslední iterace nedokázala vypočíst žádný ze smazaných bitů, nebo tím že počet iterací dosáhl nastaveného maxima.

3.1.8 bfDecoder

bfDecoder je třída deklarovaná ve hlavičkovém souboru **ldpcdecoder.h**, která reprezentuje LDPC dekodér na BSC neboli Bit-flip kanálu. Instance této třídy je schopna po zadání kódu dekódovat zprávy přijaté po přenosu BSC kanálem a opravovat v nich chyby vzniklé šumem. Pro dekódování využívá klasického algoritmu rozhodování většinou, který je uveden výše.

```
std::vector<bin> decode(std::vector<int> input,
const int maxIter = NOTSET)
```

Slouží k dekódování kódované zprávy zprávy, která je vektorem typů bin. Protože nevíme zda při přenosu došlo k chybě, bude za opravenou zprávu považována taková zpráva, jejíž všechny rovnice na kontrolních vrcholech Tannerova grafu vycházejí sudě. Hodnota maxIter je-li nastavena udává maximální počet iterací, kterými proces dekódování může projít. Dekódování skončí buď tím, že zpráva byla celá opravena, nebo tím že poslední iterace neprovedla na zprávě žádnou změnu, nebo tím že počet iterací dosáhl nastaveného maxima.

3.1.9 spDecoder

spDecoder je třída deklarovaná ve hlavičkovém souboru **ldpcdecoder.h**, která reprezentuje LDPC dekodér na BSC neboli Bit-flip kanálu. Instance této třídy je schopna po zadání kódu dekódovat zprávy přijaté po přenosu BSC kanálem a opravovat v nich chyby vzniklé šumem. Pro dekódování využívá algoritmu Sum-Product, který je uveden výše.

```
spDecoder(double errorProbability)
```

Konstruktor tohoto dekodéru vyžaduje hodnotu předpokládané pravděpodobnosti otočení bitu,tedy pravděpodobnost výskytu chyby při přenosu.

```
std::vector<bin> decode(std::vector<int> input,
const int maxIter = NOTSET)
```

Slouží k dekódování kódované zprávy zprávy, která je vektorem typů bin. Protože nevíme zda při přenosu došlo k chybě, bude za opravenou zprávu považována taková zpráva, jejíž všechny rovnice na kontrolních vrcholech Tannerova grafu vycházejí sudě. Hodnota maxIter je-li nastavena udává maximální počet iterací, kterými proces dekódování může projít. Dekódování skončí buď tím, že zpráva byla celá opravena, nebo tím že poslední iterace neprovedla na zprávě žádnou změnu, nebo tím že počet iterací dosáhl nastaveného maxima.

3.1.10 Další funkce a třídy

Další hlavičkové soubory,které jsou součástí práce slouží k simulaci a testování. Tyto třídy byly přidány pro možnost demonstrace LDPC kódování v simulovaném prostředí.

3.1.10.1 BECChannel a BFChannel Hlavičkový soubor **channels.h** obsahuje třídy BECChannel a BFChannel pro simulaci kanálu daného typu. Pomocí nastavení chybovosti zde lze simulovat šum a získat tak poškozenou zprávu. Obě třídy implementují metodu transferRandom, která vrací poškozenou zprávu.

Poznámka 3.1 BECChannel vrací touto metodou vektor typů int, aby odlišil smazané bity, které vyplňuje jako -1, pro dekodér ale stačí bude li na pozici smazaného bitu libovolné číslo kromě 0 a 1.

3.1.10.2 converter.h Obsahuje funkce pro příklad převodu dat na vektor typů bin a opačně.

```
std::vector<bin> strToBin(std::string input);
std::string binToStr(std::vector<bin> input);
```

Pro převod celých čísel.

```
std::vector<bin> intToBin(int input);
int binToInt(std::vector<bin> input);
```

Pro převod řetězců.

```
std::vector<bin> fileToBin(std::string input);
void binToFile(std::vector<bin> input, std::string output);
```

Pro převod souborů daných cestou v parametru input/output.

3.2 Ukázkový program

Následuje příklad využití implementace LDPC kódů ve sdílené knihovně AmhorA v ukázkovém programu. Tento program provede simulaci přenosu souboru zarušeným BSC kanálem a jeho opravu pomocí standardního LDPC kódu. Dekódování se provede algoritmem Sum-Product. Program LDPCshow, který je součástí přílohy na CD, se spoští se čtyřmi parametry.

- Prnvním parametrem je cesta k LDPC matici. Ukázkové matice jsou přiloženy ve složce LDPCshow/matrices/.
- Druhým parametrem je název souboru, který bude přenášen.
- Třetím parametrem je pravděpodobnost výskytu chyby na jednom bitu zprávy uváděná v promile.
- Čtvrtým parametrem je maximální počet iterací dekodéru.

Výsledkem práce programu jsou dva soubory, jejichž název vznikne přidáním přípony za název přenášeného souboru.

- Soubor s příponou .dec je soubor, který prošel kódováním, rušením a dekódováním.
- Soubor s příponou .dam je soubor, který prošel kódováním a zarušením, a slouží pro demonstraci souboru před opravou dekodérem.

Pro lepší ukázku funkce programu je lepší používat soubory s žádným, či minimálním formátováním, aby bylo možné soubor otevřít i v případě, že bude poškozen. Například jednoduché neformátované textové soubory, nebo jednoduché rastrové obrázky (např. ppm). Pro spuštění programu bez rušení stačí zadat nulu do třetího parametru, pro spuštění bez opravy zprávy stačí zadat nulu do čtvrtého parametru.

3.2.0.3 Vysvětlení práce programu

Po načtení parametrů sestavíme LDPC kód zavoláním metody construct() na objekt lcon třídy ldpcConstructor. Odtud pak získáme cesty k souborům s upraveným kódem pro kodér a dekodér, zde požádáme o kodér přímo objekt lcon. Po načtení souboru do binárního vektoru je nutné počítat s délkou akceptovaného slova kodérem a natáhnout zprávu na požadovanou délku, a tento prefix pak na konci zase odstraníme. V cyklu se pak celá zpráva rozděluje na menší celky, které jsou postupně kódovány, zarušeny a dekódovány. Výsledky se ukládají do výstupních vektorů, a nakonec se zase zapíší do souborů. Na standardní výstup se vypíše statistika chyb nasbíraná během procesu.

3.2.0.4 Ukázka použití programu Pro ukázku uvedu výsledek programu pro malý obrázek ve formátu ppm, při parametru rušení[2] 2, maximálním počtu iterací 20 a s LDPC kódem 600.1400.3, který je součástí přílohy. Celkový počet chybných bitů při odeslání byl 836, z toho 616 bylo v části s daty, a zbytek chyb se vyskytoval v kontrolních bitech. Z těchto 616 chyb zbylo po opravě 242, a chyby které přetrvaly po opravě v kontrolní části byly odříznuty.

Obrázek 5: Původní obrázek

Obrázek 6: Po zarušení kanálem

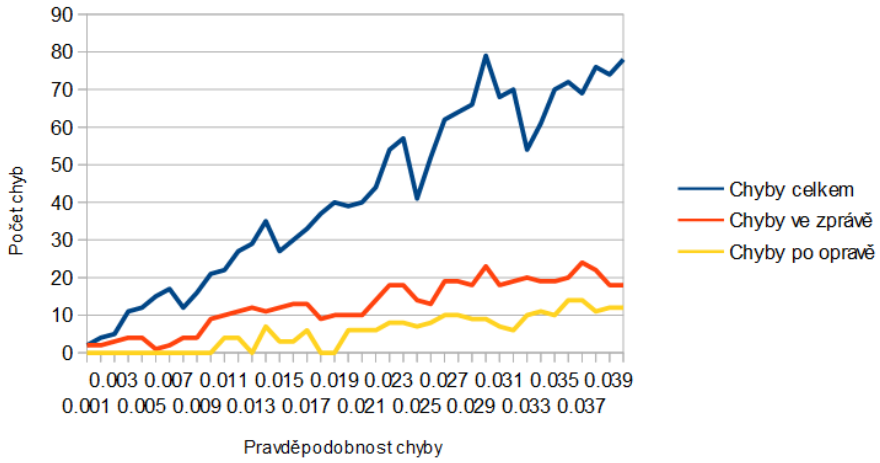
Obrázek 7: Po opravě dekodérem

[2] Při volbě větší míry zarušení by mohlo hrozit, že dojde k poškození hlavičky ppm souboru a poruší se tak jeho formát. V takovém případě bychom u poškozeného souboru museli opravit hlavičku ručně, abychom si mohli poškozený obrázek prohlédnout.

4 Simulace implementovaných algoritmů

4.1 Simulace přenosu na BEC kanálu

Tento test byl proveden standardním kódováním. Na náhodných zprávách procházejících simulovaným kanálem tvořeným třídou BECChannel se vzrůstající pravděpodobností chyby. Protože kódovaná zpráva je větší, než zpráva samotná, je navíc uváděn údaj, který objasňuje kolik chyb zasahovalo do zprávy samotné. LDPC kódem je v tomto případě kód 1920.1280.3.303, který je v příloze. Výsledek simulace byl zaznamenán v následujícím grafu:



Obrázek 8: Výsledky simulace H-D algoritmu na BEC kanálu.

Osa x grafu udává pravděpodobnost vymazání bitu při průchodu BEC kanálem. Modrá čára reprezentuje celkový počet bitů z celkových 1920, které byly smazány během průchodu kanálem. Červená čára udává počet smazaných bitů v prvních 1280 bitech, tedy v té části, která je původní zprávou. Žlutá čára udává kolik zbylo smazaných bitů po zpracování dekodérem. Toto číslo nemůže přesáhnout počet smazaných bitů ve zprávě, protože BEC dekódování nemůže zprávu poškodit více, než už je, navíc protože všechny smazané bity, které se nepodařilo rozluštit, dostanou náhodnou hodnotu. Maximální počet iterací byl sice nastaven na 30, ale ve skutečnosti nepřesáhl 3.

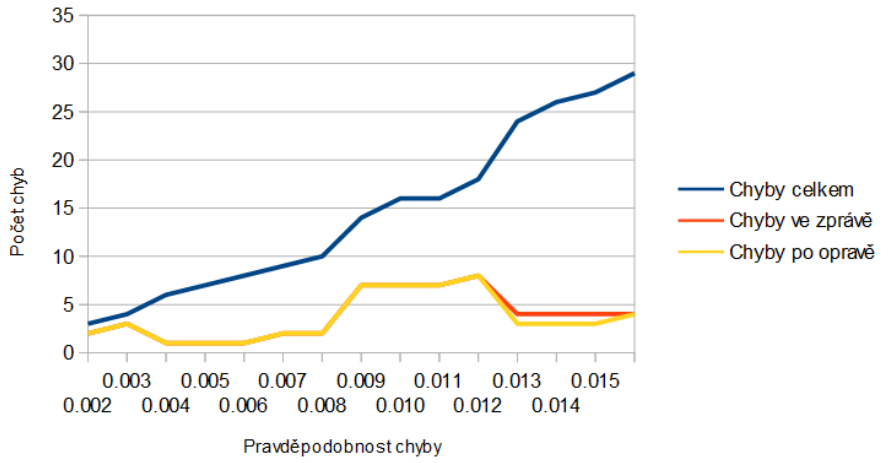
Je zřejmé, že když budeme pokračovat v simulaci dále, budou se počty chyb zbylých po opravě stále pohybovat kolem poloviny počtů ve zprávě před opravou. Je to proto, že čím více budou přibývat chyby, tím více bude algoritmus neschopen vypočítat hodnotu smazaných bitů. Těmto bitům pak musí být přiřazena náhodná hodnota ,a je zřejmé, že pravděpodobnost přiřazení správné hodnoty každému smazanému bitu je 1:2. Použitý kód se po převedení na generující a dekódující matici stává "neřídkým", váhy řádků a sloupců jsou vysoké, což ústí v příliš provázané rovnice, které v sobě pak spojují více

smazaných bitů. Lepších výsledků bychom možná dosáhli kódováním s převodem na dolní trojúhelníkový tvar.

4.2 Simulace přenosu na BF kanálu

4.2.1 Standardní hard-decision algoritmus

Tento test byl opět proveden standardním kódováním. Na náhodných zprávách procházejících simulovaným kanálem tvořeným třídou BFChannel se vzrůstající pravděpodobností chyby. LDPC kódem je v tomto případě kód 800.1400.3, který je v příloze. Výsledek simulace byl zaznamenán v následujícím grafu:

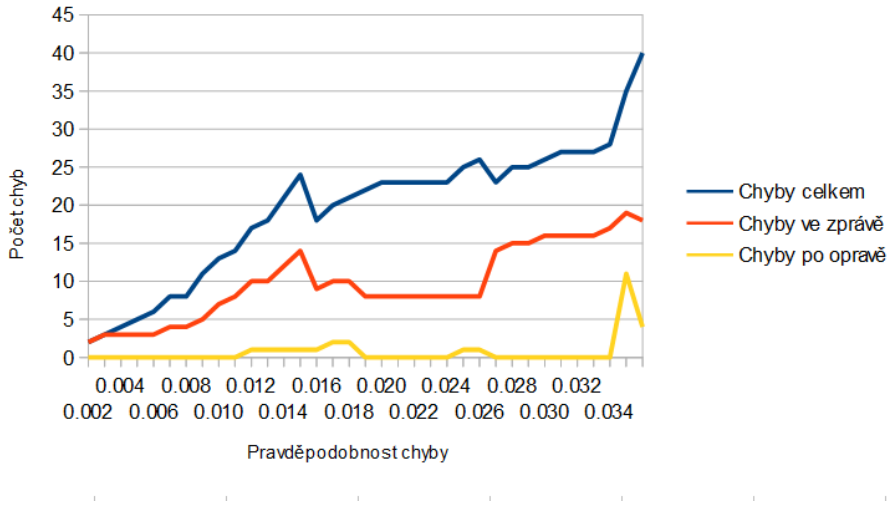


Obrázek 9: Výsledky simulace na BF kanálu (H-D algoritmus).

Jak je vidět, není tento algoritmus příliš efektivní, což jsme si ostatně ukázali už dříve. Kdybychom pokračovali v simulaci dál, začal by algoritmus zprávu dokonce ještě více poškozovat, protože nelze vědět, které bity byly přenosem otočeny. Jak jsme si ukázali dříve, vadný bit otočí hodnotu všem bitům, se kterými sdílí kontrolní vrchol, a pokud neexistuje dostatek kontrolních vrcholů, kde tyto bity nabudou správné hodnoty, zůstanou tyto bity po dekódování otočené.

4.2.2 Sum-product algoritmus

Tento test byl opět proveden standardním kódováním. Na kanálech stejného typu jako v předchozím případě, znovu se vzrůstající pravděpodobností chyby. LDPC kódem je v tomto případě kód 600.1400.3, který je v příloze. Výsledek simulace byl zaznamenán v následujícím grafu:

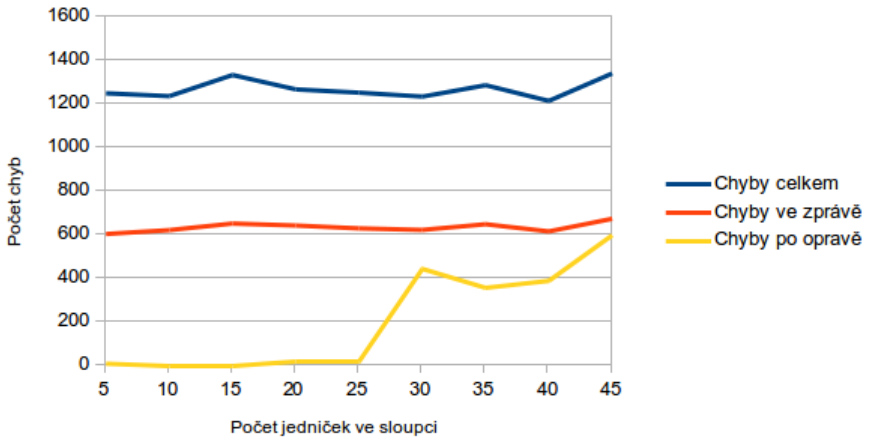


Obrázek 10: Výsledky simulace na BF kanálu (S-P algoritmus).

Tento algoritmus má mnohem lepší výsledky, než standardní hard-decision algoritmus, přesto i zde může po dalším pokračování očekávat, že bude po dekódování zpráva poškozena ještě více. Navíc je třeba zmínit, že musíme znát pravděpodobnost otočení bitu pro vytvoření dekodéru, a této pravděpodobnosti se jsme schopni (podle pravděpodobnosti na kanálu) hodně přiblížit.

4.3 Vliv řídkosti matice

V této simulaci zkusím prezentovat, jakým způsobem ovlivní počet jedniček schopnost dekodéru opravovat porušenou zprávu. Matice, na které simulace proběhla má 600 řádků a 1200 sloupců, tedy kódovaná zpráva je dvojnásobně větší oproti původní zprávě. Počet jedniček bude proměnlivý a abychom zajistili, že výsledná matice bude skutečně LDPC maticí, bude tato tvořena z levé části jednotkovou maticí 600×600 a z pravé části sloupcově regulární maticí 600×600. Zarušení v této simulaci proběhlo na bit-flip kanálu s pravděpodobností chyby $0,2$, a zvolený dekódovací algoritmus byl Sum-product algoritmus s maximálním počtem iterací 50. Celkový počet odeslaných bitů datové části (bez kontrolních bitů) byl 31800.



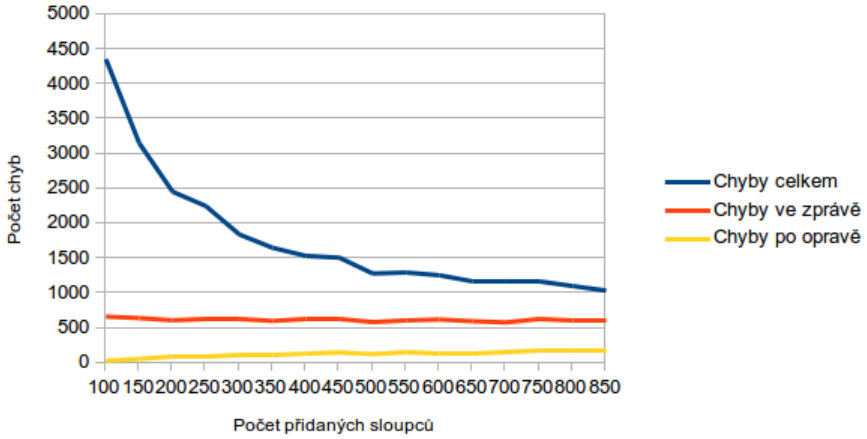
Obrázek 11: Simulace proměnlivé řídkosti matice

Jak ukazuje výsledný graf 11 od určité meze se s klesající řídkostí matice rapidně snižuje schopnost dekodéru opravovat kód. S vzrůstajícím počtem jedniček v matici vznikají komplexnější vazby mezi bitovými a kontrolními vrcholy, což vede ke kontrolním rovnicím s příliš mnoha prohozenými bity na to, aby dokázaly určit jejich správné hodnoty. V praxi se nejčastěji používají matice, jejichž maximální váha sloupce nepřesáhne 3.

4.4 Vliv počtu sloupců

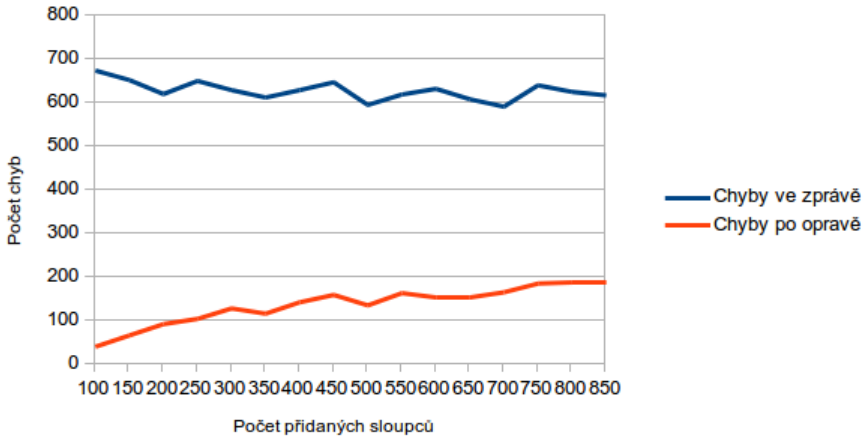
V této simulaci si ukážeme, jakým způsobem ovlivní počet sloupců matice proces dekódování a přenášenou zprávu samotnou. Matice, na které simulace proběhla má opět 600 řádků a proměnlivý počet sloupců, opět bude tvořena z levé části jednotkovou maticí 600×600 a z pravé části sloupcově regulární maticí $600 \times x$, kde x je počet přidaných sloupců. Je zřejmé že x musí být větší než 0, jinak bychom přenášeli pouze původní zprávu samotnou. Konstantní váha těchto přidaných sloupců byla 3. Zarušení v této simulaci proběhlo na BEC kanálu s pravděpodobností chyby $0,2$, a zvolený dekódovací algoritmus byl výše uvedený standardní hard-decision algoritmus s maximálním počtem iterací 30. Celkový počet odeslaných bitů datové části (bez kontrolních bitů) byl přibližně[3] 31800.

[3]Protože se mění počet sloupců, mění se i délka akceptované zprávy u kodéru. Takže počet bitů datové části musel být vždy upraven tak, aby se přizpůsobil kodéru. Ve skutečnosti však tato odchylka dosahovala maximálně několika set, což je počet ve srovnání s celkovou délkou zprávy zanedbatelný



Obrázek 12: Simulace proměnlivého počtu sloupců matice

Jak ukazuje graf 12 dochází s přidáváním sloupců do matice k výraznému poklesu chyb v celkové zprávě. Tento fakt je způsoben tím, že po transponování pravé části matice pro kodér například u matice, kde počet přidaných sloupců je roven 100, tedy matice o rozměrech 600×700, získáme matici o rozměrech 100×700. Je třeba si uvědomit že kodér v takovém případě přijímá zprávy o délce 100 bitů a vrací kódované zprávy o délce 700 bitů, původní zprávu tedy "natáhne"sedmkrát. Vysoký počet chyb v celkové zprávě je tedy způsoben pouze větší délkou zprávy samotné. S přibývajícím počtem sloupců původní LDPC matice pak klesá délka kódovaných zpráv a tím i celkový počet chyb. Jak se projeví tento fakt na schopnosti dekodéru opravovat chyby lépe znázorní graf 13.



Obrázek 13: Simulace proměnlivého počtu sloupců matice

Jak lze očekávat, s přibývajícím počtem sloupců propouští dekodér stále více chyb, protože má k dispozici stále méně kontrolních bitů. Nízkým počtem sloupců v LDPC matici jsme tedy schopni dosáhnout lepších výsledků při dekódování, avšak za cenu delších zpráv při přenosu a tím i vyšším zatížením přenosového kanálu a nejspíš i zpomalení přenosu.

5 Závěr

Pomocí LDPC kódů jsme schopni do jisté míry opravit zarušenou zprávu. Efektivita této rekonstrukce záleží, jak jsme si ukázali nejen na kódu, ale také na použitém algoritmu a na použitém kanálu. Implementace byla otestována na výše uvedených maticových souborech. Délka doby zpracování závisí na velikosti matice a možném počtu iterací při dekódování. Pro urychlení inicializace kodéru a dekodéru se upravené matice ukládají do souborů a je tak možné je načíst přímo z těchto souborů. Implementace těchto kódů je k dispozici v knihovně AmphorA na přiloženém CD spolu s programem LDPCshow, který byl vytvořen pro demonstraci jejich použití. Jako možnost dalšího rozšíření práce se nabízí možnost impplementace prvků multiplexer a demultiplexer, a s nimi též implementace simulace typů kanálů s jakými se můžeme setkat při přenosu v praxi.

6 Reference

[1] JOHNSON, S. J. *Introducing Low-Density Parity-Check Codes*. University of Newcastle, Australia, 2006, V1.1

[2] MCKAY D. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2005, V7.2

[3] GALLAGER R. G. *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

7 Přílohy

I. CD zde je uložena knihovna AmphorA a ukázkový program LDPCshow.